# Robust Software Agents with the Jadescript Programming Language

Giuseppe Petrosino[1,†], Stefania Monica[1,†] and Federico Bergenti[2,*,†]

[1] *Dipartimento di Scienze e Metodi dell'Ingegneria, Università degli Studi di Modena e Reggio Emilia, 42122 Reggio Emilia, Italy*

[2] *Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università degli Studi di Parma, 43124 Parma, Italy*

Abstract

This paper discusses several recent additions to the Jadescript agent-oriented programming language that regard the effective detection and handling of exceptional and erroneous situations at runtime. These new features were introduced to better support the mission-critical level of robustness that software agents are normally demanded to exhibit. The description of these new features is supported by an analysis of the state of the art of exception handling in programming languages, and it is complemented by a discussion on planned future developments. First, the novel exception handling mechanism introduced in Jadescript is presented, and the conceptual similarities and differences with the exception handling mechanisms normally provided by mainstream programming languages are emphasized. Second, the recent additions to Jadescript designed to support failures in behaviours are described, and these additions are related to the novel exception handling mechanism. Finally, the recent language support to manage stale messages using dedicated message handlers is presented and discussed.

**Keywords**

Robust software agents, exception and error handling, Jadescript, JADE

## 1. Introduction

Agents and *Multi Agent Systems* (*MASs*) provide a good candidate paradigm to create robust software systems. For example, agents can react to perturbations in the environment, but they can also exhibit proactivity, acting timely to prevent problematic situations. On the other hand, MASs can benefit from the fault-tolerance mechanisms and strategies that have been already studied in distributed computing, e.g., replicating services and gracefully degrading the system to a state of limited functionality.

In general, exceptions are used in software development to represent particular situations that can be encountered at runtime. In agent technologies, these situations can arise from the inner machinery of each individual agent or from the obstacles that it encounters while trying to achieve its design goals. However, in MASs, exceptional or erroneous situations can also arise in the social structures that are created among interacting agents.

*WOA 2022: 23rd Workshop From Objects to Agents, September 1–2, Genova, Italy*

*Corresponding author.

†These authors contributed equally.

giuseppe.petrosino@unimore.it (G. Petrosino); stefania.monica@unimore.it (S. Monica); federico.bergenti@unipr.it (F. Bergenti)

0000-0001-6234-5328 (G. Petrosino); 0000-0001-6254-4765 (S. Monica); 0000-0002-4756-4765 (F. Bergenti)



All these challenges and opportunities are tackled by software engineering techniques, and they can also be effectively addressed by means of tools integrated in the languages in which software agents are written. These techniques commonly fall under the umbrella of exception handling, and the set of tools in charge of helping programmers to achieve exception handling are normally called exception handling mechanisms. These mechanisms are specifically required for *Agent-Oriented Programming* (*AOP*) [1] languages, and they cannot be limited to the mechanisms adopted by mainstream programming languages.

Jadescript [2, 3, 4, 5] is a recent AOP language designed to develop JADE [6, 5] agents. It provides a set of agent-oriented linguistic constructs and related abstractions, namely agents, (agent) behaviours, and (communication) ontologies. Agents written in Jadescript are executed in JADE platforms, and they interact via asynchronous messaging. Therefore, Jadescript adopts an event-driven programming style. It presents declarative aspects like pattern matching [7], but the language is mostly imperative, and it adopts a syntax that recalls the syntax of agent pseudocode. As an AOP language, Jadescript requires specific language facilities to represent and handle exceptions, and it offers original solutions to problems pertaining to this scope. Note that the solutions that Jadescript offers satisfy a set of key requirements:

1. They must be useful, i.e., they must effectively help programmers with representation of exceptional situations and detection/prevention of errors;
2. They must to properly blend with the existing constructs of the language;
3. They must not tamper with desirable properties of the source code of Jadescript agents, like readability, modifiability, and reusability;
4. They must match the programming style adopted by Jadescript (in particular, event-driven programming); and
5. They must not affect the runtime performance of nominal behaviours (i.e., behaviours in the absence of perturbations and exceptions).

This paper presents three new language facilities for exception handling that extend Jadescript and that satisfy these requirements.

Moreover, this paper includes in Sect. 2 an overview of the state of the art on exception handling mechanisms in software agents and MASs. The overview provides relevant background information on exception handling in agents, and it also includes a discussion on exception handling mechanisms in MASs. These mechanisms treat exception handling as a problem of society design, whose solution involves social and coordination abilities. However, the main contribution of this paper concerns agent design, and societal issues are not treated for the following reasons. First, no mechanisms to represent and handle exceptional situations related to the internal errors of agents were available in Jadescript. The added improvements to the language are described in detail in Sect. 3. Second, the presented improvements are also related to mechanisms that are strongly related to the Jadescript agent model. These mechanisms are called behaviour failure and stale message handler, and they are discussed in Sect. 4 and Sect. 5, respectively. Finally, Jadescript can possibly include a support for explicit exception handling at the MAS level, but such a support would require language features (e.g., interaction protocols) that are still planned for future developments. These and other future developments are briefly discussed in Sect. 6, which also concludes the paper.

## 2. Related Work

Exception handling in software systems was extensively studied, e.g., in [8, 9, 10]. These papers mostly focus on the main design issues of structured exception handling for traditional programming languages. Actually, these papers define exception conditions as conditions, detected while attempting to perform an operation, that are brought to the attention of the operation user by the operation itself.

Such a relationship between operation users and operations is apparent in Akka [11], which proposes the supervision model as the core mechanism to handle exception conditions in the actor model. Akka actors are organized in a hierarchy characterized by dependency relationships among actors and their creators (parents). When an actor $a_j$ creates the actor $a_k$, $a_j$ is said to be the supervisor of $a_k$, which, in turn, is said to be its subordinate. Supervisors request the completion of sub-tasks to subordinates, and in turn handle their failures. Each time an actor fails, its supervisor is notified and it intervenes by:

1. Aborting the execution of the subordinate actor;
2. Resuming the execution of the subordinate actor, removing the conditions that caused the failure;
3. Resetting the subordinate actor, and therefore restarting its execution with a fresh internal state; or
4. Failing, thus escalating the failure.

Notably, this approach promotes the *let it crash* fault tolerance model, in which faulty actors are designed to be destroyed to let supervisors prepare new attempts at completing the operations.

SARL [12] is both an AOP and an object-oriented programming language. As a member of the second category, it proposes an exception handling mechanism similar to the ones used in mainstream object-oriented programming languages. Exception throwing is performed by means of the usual `throw` statement, and `try/catch/finally` is used to handle exceptions at runtime in a structured way. Note that SARL is based on Xtend [13], which is a Java dialect. For this reason, SARL agents are executed by the Java virtual machine, and the feedback about the exception conditions is contained in objects whose classes implement the `Throwable` interface. In addition, SARL proposes a supervisor mechanism similar to the mechanism that Akka uses. This mechanism is based on the possibility of failure events to be propagated to parent agents by means of the builtin `emitToParent` function.

In addition, Baldoni et al., in their recent paper [14], showed how SARL can be extended to achieve a richer sort of exception handling to target the MAS level. Their proposal allows to create social expectations with respect to exception handling in the conducts of agents. This extension can be implemented with the adoption of a special SARL space, in which each agent can emit events to:

1. Register as handler for a specific type of exception;
2. Register as potential raiser of a specific type of exception, as a consequence of a particular type of event; or
3. Raise exceptions, represented by means of Java objects.

The proposed special SARL space provides a mechanism used by raiser agents to notify handler agents about exceptional situations, but it also provides a way to create awareness about the distribution of responsibility on exception handling among interested agents.

A counterargument to the supervisor approach is that it limits the degree of autonomy of agents. This is one of the main reasons that drove Platon et al. [15, 16] to propose an exception handling mechanism that keeps the autonomy of agents as an aspect of primary importance. Their proposal is based on the idea that *"*[*agents*] *can take their own decisions, notably concerning normal and exceptional events"*, and therefore *"exceptions make sense inside agents"* [17]. Their proposal is based on a specific agent model (the exception-ready agent) that lets agents detect exceptions in their interactions with the environment. An exception-ready agent evaluates percepts on the basis of relevance and expectations, using an internal representation that is updated as the agent acts on the environment. From each percept evaluation, an agent can automatically classify an event as normal or exceptional. Events belonging the latter class are handled by application-specific mechanisms internal to the agent.

Similarly, Souchon et al. [18] claim that supervisor-type architectures are not a satisfactory solution for MASs because agents should encapsulate their own conducts. They combine this argument with the idea of the need for an exception handling mechanism that should be integrated with the agent platform, and they implemented this view in their Sage model for the MadKit MAS. Their model is based on their vision of agents as service providers and as role takers. Therefore, they implemented an exception handling mechanism that supports concerted exception handling at the service and role levels, in which agents can coordinate to handle critical situations represented by sets of exceptions signaled from the called services or from the responses of the agents with specific roles.

In the *Belief-Desire-Intention* (*BDI*) AOP language Jason [19, 20] exceptional situations in plan execution are managed by means of the plan failure feature. In particular, it is assumed that a plan in the intention stack of the agent triggered by an event `+!g` can always fail. When this happens, the Jason interpreter automatically puts a `-!g` goal on top of the same intention stack. This causes the agent to search for an appropriate plan in its plan library to be executed as an alternative. The approach that Jason proposes allows programmers to code robust reactions, e.g., to reattempt to reach the goals with an alternative course of actions (i.e., contingency plan), or to create a new set of conditions for which the original plan is expected to succeed. Jason does not provide a mechanism to explicitly program agents that coordinate to handle exceptions at the MAS level. However, JaCaMo [21], which is the MAS development suite composed of Jason, CArtAgO [22], and Moise [23], provides effective exception handling through responsibility distribution in agent organizations [24].

JADE, and therefore Jadescript, already provides ways to represent exceptional and erroneous conditions in the MAS. Actually, as specified by FIPA [25], there are several performatives that can be used to communicate the presence of exception conditions. In particular, `not_understood` messages are used to signal message validation errors, i.e., to signal that an unexpected or malformed message was received. Moreover, `failure` messages can be used to signal failed attempts at performing actions. Messages with the `failure` performative are required to provide (in a tuple, together with the attempted action) a proposition to describe the reasons for which the attempted action failed. Note that a `failure` message always implies that the attempted action was considered feasible by the agent at the time it attempted to perform

it. Finally, an agent can signal to a requester agent that a particular requested action is not supported or not authorized by means of a `refuse` message. The content of this message is always a tuple containing the action that was not performed and a proposition describing the reason for the refusal. These messages are used by Jadescript agents during their interactions, but they are of primary importance also to perform core operations in the agent platform. This is because the *Agent Management System* (*AMS*) of a JADE platform provides essential services (e.g., agent creation) that are supposed to be accessed by means of `request` messages. Therefore, to handle exceptions during the execution of these operations, the agent is supposed to be able to handle and interpret `not_understood`, `failure`, and `refuse` messages.

## 3. Exceptions in Jadescript

Jadescript descends from JADEL [26], and it compiles to Java [27]. JADEL was based on a dialect of Java, namely Xtend [13], which supports Java exceptions with the `throw` statement and the ordinary `try/catch/finally` construct. All these exception-related facilities were inherited by JADEL from Xtend, but their immediate inclusion in Jadescript required attentive selection and ultimately called for a deep redesign.

First, it was decided not to include Java-like checked exceptions in Jadescript. These exceptions represent a good tool to increase software robustness by preventing exceptions to occur without anticipation in the context of a method invocation. As a matter of fact, checked exceptions are a subset of exceptions that, in order to be thrown by a method, require to be explicitly declared in the signature of the method. However, checked exceptions come with several drawbacks. Notoriously, checked exceptions tamper with modifiability because existing code cannot be easily updated to treat new exception conditions. For example, if a method declares that it can throw exceptions of type `E1`, and in a later version of the code it needs to address scenarios in which `E2` exceptions, unrelated to `E1` exceptions, are thrown, the introduction of this change propagates to all the uses of the method. This kind of refactoring is considered acceptable by someone, arguing that the change of the signature of a method is an operation that is often done by simply changing its name, argument count and types, and return type. However, these features of methods tend to be more unchanging than exceptions, whereas exceptions are related to conditions that are often difficult or impossible to predict when designing the method in its first implementations. Checked exceptions force the explicit acknowledgment of these conditions in the caller code, producing friction with adaptability.

Second, it was decided not to include the `try/catch/finally` construct in Jadescript. This choice was taken because of the significant reduction of readibility that this construct introduces in the language, as the it may obfuscate control flow, increase the risk of resource leaks, and promote convoluted code. Moreover, this construct tends to be difficult to understand, as its specification *"contains a large number of corner cases that programmers often overlook"* [28]. The solution to allow exception handing in Jadescript described in this section does not use this construct, and it blends well with the other key features of the language. Most importantly, it preserves the characteristic readibility that is an appreciated quality of the language.

In Jadescript, programmers implement the details of the behaviours of their agents as procedural sections of code included in procedures, functions, and event handlers. These procedural

sections are structured sequences of operations that can fail at runtime. Examples of common operations that can fail at runtime are:

1. Division by zero;
2. Access to an nonexistent value in a list, which is referenced by an index that is not within the range of valid indexes;
3. Access to an nonexistent entry in a map, which is addressed using a key that has not been assigned to an entry; and
4. Convert a `text` into another value (e.g., `integer`, `real`, `timestamp`), when the input value is not properly formatted for the conversion.

The presence of these, and other similar failures, implies that the language requires features to effectively detect and handle exception conditions. To meet this requirement, a new exception handling feature has been recently added to Jadescript. The novel exception handling feature encompasses three aspects that are worth discussion:

1. A way to represent the exception conditions in a structured manner;
2. A way to notify the operation caller of the emergence of exception conditions; and
3. A way to detect exception conditions from the context of the operation caller.

To meet the first requirement, the new exception handling mechanism uses propositions. This design choice is particularly fit for Jadescript because it already provides the proposition abstraction to model facts about the world, and therefore propositions can be used to describe the conditions that caused the operation to notify the caller. In particular, application-specific atomic propositions and predicates can be declared by means of `proposition` and `predicate` declarations, respectively, in user-defined ontologies. For built-in exceptions and errors, the corresponding propositions and predicates are predefined in the provided basic ontology. For example, accessing a list with an invalid index would produce an exception described by the `IndexOutOfBounds` proposition. Please note that the basic ontology is used by all Jadescript agents and its concepts, actions, and propositions are always available to programmers.

Exceptions can be notified using the `throw` statement. As in other ordinary procedural languages, the statement is introduced by the `throw` keyword, and it requires an expression. In Jadescript, this expression must evaluate to a proposition describing the exception condition.

The on `exception` event handler was added to the language to detect exceptions. Several occurrences of this event handler can be added to the body of agents and behaviours. By default, an on `exception` handler captures all kinds of exceptions, but several constraints can be added to the head of the handler to allow programmers to restrict the set of captured exceptions. As for message handlers, the constraints on the captured exceptions can be expressed in two ways, which can be combined. The first way, called `when`-expression, declares constraints written as Boolean expressions that can include values extracted from the exception and from the state of the agent. Note that `when`-expressions must be composed of operations without side effects, and this requirement is checked at compile-time. The second way, called pattern matching, is already available for message handlers [7], and it can be used to easily declare the type and the internal structure of the proposition describing the expected exception, destructuring the proposition in parts. The proposition describing the exception condition can be accessed in handler bodies and in `when`-expressions by means of the `exception` keyword.

```
agent AlarmClock
  property wakeUpTime as timestamp

  # Receives a text (time) containing the timestamp as start-up argument
  on create with time as text do
    # Attempts to convert time to a timestamp, throwing an exception if the
    # format of time is invalid
    wakeUpTime of this = time as timestamp
    # Activates a behaviour performing the task at the specified time
    activate DoWakeUp at wakeUpTime

  # Exception handler that aborts the agent because the start-up argument
  # is invalid
  on exception IvalidTimestampTextFormat(theText) do
    log "Invalid timestamp format for argument '" + theText + "'."
    do shutdown
```

**Figure 1:** An example of an exception handler used to capture invalid arguments passed to an `AlarmClock` agent.

When multiple exception handlers are defined within the same behaviour declaration or agent declaration, the handler to be executed is selected by attempting to match the exception description against the pattern and the `when`-expression of each exception handler. The search continues until a handler matches successfully, but when no applicable handlers can be found, the exception escalates. An exception escalation results in two different outcomes, depending on the context. If the exception escalates from the code of an agent declaration, the agent shuts down after emitting an appropriate message to the log and properly deregistering from the platform. Conversely, when the exception escalates from the code of a behaviour declaration, the behaviour fails. Fig. 1 shows a simple example of an `on exception` handler that uses a pattern to capture relevant exceptions.

## 4. Behaviour Failure

In Jadescript and in JADE, (agent) behaviours are the main components of the layered architecture of the agents. They are executed concurrently by the agent, and they are used to model the tasks that are performed by the agent. The operations that compose these tasks can fail for several reasons, like the ones described in Sect. 3. Moreover, some scenarios can benefit from the explicit signaling of the failure of a behaviour as part of the design of the agent. Note that a well-designed behaviour can fail even if no operations threw an exception. For example, a behaviour can detect, during its execution, that the requested task could no longer be completed. In these cases, the failure of the behaviour correctly expresses the general idea that some conditions prevent the task to be successfully performed.

Note that the failure of a behaviour for the reasons mentioned above is different from the action of throwing an exception. The latter simply indicates that, during the execution of an

operation, some conditions that require attention are met. The failure of a behaviour, instead, brings an additional negative meaning (i.e., the task can no longer be completed). Moreover, the exceptions of ordinary procedural languages always happen in a context in which there is a hierarchy of operations defined by the caller-called relationships. Therefore, when an exception occurs, a responsibility chain can be computed by navigating the call stack from top to bottom. Behaviours do not enjoy such a hierarchical structure, since, even if a behaviour can (create and) activate another behaviour, both behaviours continue to execute concurrently in the agent. In addition, any behaviour can be deactivated and reactivated at a later time by a different behaviour. For these reasons, it is useful to have an exception handling mechanism that is intimately designed around the idea of behaviour failure and that matches the peculiarities of the behaviour execution model in Jadescript agents. This new mechanism encompasses two ways to cause the failure of a behaviour:

1. An unhandled exception escalates from the code of the behaviour; and
2. The `fail` statement is executed for a specific behaviour.

Sect. 3 describes the way an exception can cause a behaviour to fail in the first case. Instead, to explicitly cause a behaviour to fail in the second case, programmers can use the new `fail` statement. The syntax of the `fail` statement is the following:

⟨*FailStatement*⟩ ::= '`fail`' ⟨*Expr:Behaviour*⟩ '`for`' ⟨*Expr:Proposition*⟩

The statement requires two arguments that are provided as expressions. The first argument, which follows the `fail` keyword, is the value representing the behaviour that is failing. The second argument, introduced by the keyword `for`, is a proposition that represents the reason for the failure of the behaviour. Note that in the case of an escalated exception, the proposition describing exception condition is automatically used as the reason for the failure. Moreover, note that the expression evaluating to the failing behaviour can refer to any behaviour. This can be used to design behaviours that can detect the failure of other behaviours, thus providing a high degree of flexibility in managing failures. When a behaviour fails, it is automatically deactivated, and an appropriate `on behaviour failure` handler is executed, if available. The `on behaviour failure` handler is similar to exception handlers. Its head supports the specification of a pattern that can possibly match the proposition describing the reason for the behaviour failure. Moreover, it has a special variable `behaviour` that can be accessed within the `when`-expressions and that can be matched against behaviour patterns. Behaviour patterns are a novel addition to the language. They match against behaviours, and they are composed of two parts. The syntax of a behaviour pattern is:

⟨*BehaviourPattern*⟩ ::= (⟨*PatternVar*⟩ '`as`')? ⟨*PatternBehaviourType*⟩

⟨*PatternVar*⟩ ::= ⟨*Identifier*⟩ | '_'

⟨*PatternBehaviourType*⟩ ::= ⟨*Type:Behaviour*⟩ | '_'

where ⟨*PatternVar*⟩ is a name that unifies with the behaviour value, and ⟨*PatternBehaviourType*⟩ is the expected behaviour type [29]. Both parts can be replaced with underscore signs to include placeholders that allow any value or any type to match [29]. For example, `b as TestBehaviour`

```
agent ExampleAgent
  # The agent kills itself in case of fatal problems
  on behaviour failure FatalProblem do
    do shutdown

  # The agent activates a contingency plan, if available
  on behaviour failure when behaviour matches b as TaskWithAlternative do
    activate ContingentBehaviour(taskParameter of b)

  # The agent stubbornly reactivate failing behaviours
  on behaviour failure when behaviour matches b as _ do
    activate b
```

**Figure 2:** Examples of behaviour failure handlers with different handling strategies.

matches against any behaviour of type `TestBehaviour`, and the behaviour value is unified with the variable `b`. Another example of a behaviour pattern is `b as _`, which matches against any type of behaviour. This behaviour pattern is useful when there is the need to perform common operations on a behaviour (e.g., reactivate it) without knowing the exact type of the behaviour. Just like for the other types of event handlers, the search for a matching handler is done by checking all available handlers, from the top of the agent declaration to its bottom, until a matching handler is found. Fig. 2 shows some examples of behaviour failure handlers with different handling strategies.

## 5. Stale Message Handlers

Jadescript agents schedule active behaviours using a non-preemptive scheduling algorithm, sharing the internal state of the agent and its message inbox. The message inbox is a queue that contains all the messages received by the agent that are still waiting to be handled. Every time a behaviour is scheduled, all the headers of its `on message` handlers are checked against the messages in the inbox. This is useful to declaratively define to which incoming messages a behaviour should react. However, if a message does not match against the set of constraints of a handler, the message is kept in the inbox at its current position to ensure that another handler (potentially in another behaviour) could successfully handle it. When no handlers in any of the active behaviours successfully match against a message in the inbox, the message does not get extracted, and therefore it remains in the inbox, potentially for an indefinite amount of time (it becomes stale). This is not a problem for the execution of the agent because in JADE, and therefore in Jadescript, the inbox is a message queue with limited capacity, and when full, the oldest messages are simply discarded. This has the effect of potentially ignoring relevant messages. Actually, agents are often requested to properly react to stale messages because ignoring all of them can prevent the identification of situations that call for the attention of the agent. To do so, application-specific code is required to define proper message handlers to react to the messages that are not captured by any other handler. When the agent has only one behaviour, the processing of stale message can be easily performed by simply writing an on

```
cyclic behaviour B1 uses ontology O1
  # Handle P1
  on message inform P1 do
    # Body of the handler

  # Handle all other messages
  on message do
    send not_understood message to sender of message

cyclic behaviour B2 uses ontology O2
  # Handle P2
  on message inform P2 do
    # Body of the handler

  # Handle all other messages
  on message do
    send not_understood message to sender of message
```

**Figure 3:** A toy example that shows a simple but ineffective strategy to capture stale messages.

`message` handler at the end of the behaviour body, without constraints (i.e., with no patterns and no `when`-expressions) on the handled messages. The body of this handler can define how the agent reacts to all the incoming messages that are not expected to match against the constraints of the other message handlers. This approach is perfectly valid for single-behaviour agents, but it is problematic for agents with several active behaviours. Consider the toy example in Fig. 3. Here, two behaviours are defined, namely `B1` and `B2`. Both of them handle inform messages, with propositional contents `P1` and `P2`, respectively. Both of them also declare an `on message` handler with no constraints intended to capture all other types of messages. If both behaviours are active for the same agent, it could happen that, during the execution of `B1`, a message `inform P2` gets removed from the inbox and treated as not understood, replying to its sender using a `not_understood` performative. The opposite situation, i.e., `B2` handles a message intended for `B1`, could also happen. Moreover, note that the delegation of the task of handling stale messages to a third specific behaviour does not solve the problem. Actually, the problem of handing stale messages when agents have multiple active behaviours is not easy to solve without specific language facilities because the provision of a behaviour to check which messages are of interest for other active behaviours would be too complex and fragile.

To deal with this kinds of scenarios, two new features have been recently added to Jadescript. The first feature is the new `on stale message` handler, which extracts a stale message from the inbox and executes a section of code to handle it. To this purpose, a stale message is defined as a message that did not match against any `on message` handler after all active behaviours have been executed at least once since its reception. Consider the example in Fig. 4. In this example, the MAS contains two classes of agents (among others). The agents in these two classes, namely `Seller` and `Buyer`, provide the services for acting as intermediaries in selling and buying goods represented by the `Item` concept. In the example, only the code of `Seller` agents is

```
ontology SellAndBuy
  # An item that can be sold or bought
  concept Item(id as integer, name as text)
  # The action of selling an Item for at least minPrice
  action Sell(item as Item, minPrice as integer)
  # The action of buying an Item for at most maxPrice
  action Buy(item as Item, maxPrice as integer)

# Seller agents are designed to negotiate Sell requests
agent Seller uses ontology SellAndBuy
  on create do
    activate HandleSellRequests

# HandleSellRequests behaviours are designed to satisfy Sell requests by
# negotiating with Buyer agents
cyclic behaviour HandleSellRequests uses ontology SellAndBuy
  on message request Sell(item, minPrice) do
    # Commits to the request
    send message agree (content of message) to sender of message
    # Contacts potential buyers and starts negotiations (not shown)

  # The on stale message handler
  on stale message do
    if performative = request and content matches Buy(_,_) do
      # Action Buy is not supported by the Seller agent
      send refuse (content of message, NotSupported) to sender of message
    else do
      # In all other cases, the message could not be understood by the agent
      send not_understood (message) to sender of message
```

**Figure 4:** Simple example of a stale message handler.

shown because `Buyer` agents and their `HandleBuyRequests` behaviours are similar. When a `request Sell(item, maxPrice)` message arrives, the `Seller` agent commits to the request and it starts a negotiation (not shown in this example). Normally, a `Seller` agent does not provide the service of buying items, which is instead provided by `Buyer` agents. For this reason, the `HandleSellRequests` behaviour does not provide message handlers to capture requests to buy items. However, a `Seller` agent understands the meaning of buying and it can handle stale messages with those requests in a correct manner, i.e., by explicitly refusing the request. In these cases, the design proposed in the example strongly promotes reusability and composibility of behaviours. As a matter of fact, a `Trader` agent, defined as an agent that is both a `Seller` and a `Buyer`, can be easily implemented by activating both the `HandleSellRequests` behaviour and the `HandleBuyRequests` behaviour. The two behaviours are active at the same time, but they would not race for messages because messages are flagged as stale only if both behaviours had the opportunity to handle them at least once.

The second feature that has been recently introduced in Jadescript is the `putback` statement,

which enables a fine-grained management of the message inbox. This statement accepts a message expression as argument right after the `putback` keyword and, when executed, it puts the message in the inbox, in front of all other messages. Note that any message can be used with the `putback` statement, and it is not assumed that the message has been recently removed from the message queue.

Finally, the default behaviour of Jadescript agents in the presence of stale messages has been also changed to effectively benefit from the two recent additions to the language. When no on `stale message` handler is defined in an agent or in any of its active behaviours, the agent now automatically extracts stale messages and it handles them in the following way:

1. If the stale message is a `not_understood` message, the agent writes a message to the log to signal the problem; and
2. In all other cases, the agent replies with a `not_understood` message to the sender of the stale message.

This behaviour was adopted to make the handling of stale messages consistent with the behaviour of agents and services regulated by FIPA specifications [30], which assume that agents are designed to reply with `not_understood` messages to all messages with unexpected performative, ontology, or content.

## 6. Conclusion and Future Work

This paper presented some relevant language features that have been recently added to Jadescript. These features use dedicated language constructs that match the event-driven programming style promoted by Jadescript, keeping a high degree of readability, modifiability, and reusability without affecting performance in nominal cases. In particular, this paper discussed several additions to Jadescript that regard the representation, detection, and handling of exceptions. These exceptions, coupled with related exception handlers, can be used to detect errors and failures. Moreover, this paper shown how the failure of agent tasks, implemented by behaviours, can be tackled by means of the new behaviour failure mechanism. Finally, this paper discussed a new way to effectively define the behaviour of an agent when unexpected messages are received. In summary, the novel features that has been recently added to Jadescript, as described in this paper, can effectively help programmers to detect and handle exceptional and erroneous situations at runtime, thus achieving the level of robustness that is needed in several critical application scenarios (e.g., [31, 32]).

Future enhancements planned for the language comprehend custom interaction protocols, whose inclusion in the language provides several new opportunities regarding exception handling. Custom interaction protocols promote coordination by explicitly structuring conversations among agents as finite state machines. In this context, Jadescript agents engaging interaction protocols can consequently adopt the exception-ready agent execution model proposed by [17]. As a matter of fact, the agents that send messages act on the mental states of message receivers. These mental states have an internal representation in the sender agent, and such a representation can be abstracted from the current state of the engaged protocols, and it can be updated automatically after each message, either sent or received. Therefore, the

percepts (i.e., the received messages) can be automatically classified as normal or exceptional using a relevance filter and an expectation filter. In particular, a message can be considered as relevant if it belongs to the same conversation of the message that initiated the conversation. Moreover, a relevant message matches the expectations of the receiving agent if it belongs to the set of acceptable interactions taken from the state of the conversation according to the rules defined by the protocol. If a relevant message is classified as unexpected, the interaction protocol mechanism can throw an exception, and it can cause the failure of the behaviour designed to handle the reception of the message.

Custom interaction protocols can also be designed to support the definition of strategies for automatic recovery from erroneous states, taking inspiration from the strategies that language parsers adopt to recover from syntax errors (e.g., defining synchronization points) [33]. Interaction protocols can also be used to create stable social structures among Jadescript agents, and, in this way, they can be used to build exception raiser-handler relationships, as suggested in [14, 24]. This provides the freedom needed to distribute responsibility about exception handling among the agents participating in a protocol.

The exception handling mechanism built on the proposed behaviour failure mechanism can be improved if the language is extended to support explicit hierarchical organizations of behaviours. As a matter of fact, the current version of Jadescript includes only two types of (simple) behaviours, namely one-shot and cyclic behaviours [29]. Even if these are normally considered as sufficient to organize the tasks of the agents, Jadescript behaviours can benefit from a higher degree of reusability if the language could include composite behaviours. Composite behaviours, as available in JADE [34], are behaviours that embed child behaviours executed with specified scheduling policies (e.g., sequentially, in parallel, or based on a finite state machine). The embedder-embeddee relationships that are formed by the composition of behaviours can then be used to implement supervision strategies with respect to behaviour failures. Failures of child behaviours could be handled by behaviour failure handlers defined in parent composite behaviours, and the hierarchy tree could be traversed upward to reach the agent, which would act as the root supervisor for all of its behaviours.

## Acknowledgments

This work was partially supported by the Italian Ministry of University and Research under the PRIN 2020 grant 2020TL3X8X for the project *Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems* (*T-LADIES*).

## References

[1] Y. Shoham, Agent-oriented programming, Artificial Intelligence 60 (1993) 51–92.

[2] F. Bergenti, S. Monica, G. Petrosino, A scripting language for practical agent-oriented programming, in: Proceedings of the $8^{\text{th}}$ ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2018), ACM, 2018, pp. 62–71.

[3] F. Bergenti, G. Petrosino, Overview of a scripting language for JADE-based multi-agent systems, in: Proceedings of the 19th Workshop "From Objects to Agents" (WOA 2018), volume 2215 of *CEUR Workshop Proceedings*, RWTH Aachen, 2018, pp. 57–62.
[4] G. Petrosino, F. Bergenti, An introduction to the major features of a scripting language for JADE agents, in: Proceedings of the 17th Conference of the Italian Association for Artificial Intelligence (AI*IA 2018), volume 11298 of *Lecture Notes in Artificial Intelligence*, Springer, 2018, pp. 3–14.
[5] F. Bergenti, G. Caire, S. Monica, A. Poggi, The first twenty years of agent-based software development with JADE, Autonomous Agents and Multi-Agent Systems 34 (2020).
[6] F. Bellifemine, F. Bergenti, G. Caire, A. Poggi, JADE–A Java Agent DEvelopment Framework, in: Multi-Agent Programming, volume 25 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Springer, 2005, pp. 125–147.
[7] G. Petrosino, F. Bergenti, Extending message handlers with pattern matching in the Jadescript programming language, in: Proceedings of the 20th Workshop "From Objects to Agents" (WOA 2019), volume 2404 of *CEUR Workshop Proceedings*, RWTH Aachen, 2019, pp. 113–118.
[8] J. B. Goodenough, Exception Handling: Issues and a Proposed Notation, Communications of the ACM 18 (1975).
[9] J. B. Goodenough, Exception handling design issues, ACM SIGPLAN Notices 10 (1975).
[10] J. B. Goodenough, Structured exception handling, in: Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, 1975, pp. 204–224.
[11] Lightbend, Akka platform by lightbend, www.lightbend.com/akka-platform, 2022. Accessed on July 10th, 2022.
[12] S. Rodriguez, N. Gaud, S. Galland, SARL: A general-purpose agent-oriented programming language, in: Proceedings of the IEEE/WIC/ACM International Joint Conferences of Web Intelligence (WI 2014) and Intelligent Agent Technologies (IAT 2014), volume 3, IEEE, 2014, pp. 103–110.
[13] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing, 2013.
[14] M. Baldoni, C. Baroglio, G. Chiappino, R. Micalizio, S. Tedeschi, Exception Handling in SARL as a Responsibility Distribution, Procedia Computer Science 201 (2022) 795–800.
[15] E. Platon, S. Honiden, N. Sabouret, Challenges in exception handling in multi-agent systems, in: SELMAS 2006: Software Engineering for Multi-Agent Systems V, volume 4408 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 41–56.
[16] E. Platon, N. Sabouret, S. Honiden, A definition of exceptions in agent-oriented computing, in: ESAW 2006: Engineering Societies in the Agents World VII, volume 4457 of *Lecture Notes in Artificial Intelligence*, 2007, pp. 161–174.
[17] E. Platon, N. Sabouret, S. Honiden, An architecture for exception management in multiagent systems, International Journal of Agent-Oriented Software Engineering 2 (2008).
[18] F. Souchon, C. Urtado, S. Vauttier, A Proposition for Exception Handling in Multi-Agent Systems, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.6182&rep=rep1&type=pdf, 2008. Accessed on July 10th, 2022.
[19] R. H. Bordini, J. F. Hübner, BDI agent programming in AgentSpeak using Jason, in:

CLIMA 2005: Computational Logic in Multi-Agent Systems, volume 3900 of *Lecture Notes in Artificial Intelligence*, 2006, pp. 143–164.

[20] R. H. Bordini, J. F. Hübner, Jason: A Java-based interpreter for an extended version of AgentSpeak, jason.sourceforge.net, 2007. Accessed on July 20th, 2022.

[21] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, A. Santi, Multi-agent oriented programming with JaCaMo, Science of Computer Programming 78 (2013) 747–761.

[22] A. Ricci, M. Piunti, M. Viroli, A. Omicini, Environment programming in CArtAgO, in: Multi-agent programming, Springer, 2009, pp. 259–288.

[23] J. F. Hübner, O. Boissier, R. Kitio, A. Ricci, Instrumenting multi-agent organisations with organisational artifacts and agents, Autonomous Agents and Multi-Agent Systems 20 (2010) 369–400.

[24] M. Baldoni, C. Baroglio, S. Tedeschi, R. Micalizio, Distributing responsibilities for exception handling in JaCaMo, in: Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), International Foundation for Autonomous Agents and Multiagent Systems, 2021, pp. 1752–1754.

[25] FIPA, FIPA Communicative Act Library Specification, www.fipa.org/specs/fipa00037/SC00037J.html, 2021. Accessed on July 24th, 2021.

[26] F. Bergenti, An introduction to the JADEL programming language, in: Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2014), IEEE, 2014, pp. 974–978.

[27] G. Petrosino, E. Iotti, S. Monica, F. Bergenti, Prototypes of productivity tools for the Jadescript programming language, in: Proceedings of the 22nd Workshop "From Objects to Agents" (WOA 2021), volume 2963 of *CEUR Workshop Proceedings*, RWTH Aachen, 2021, pp. 14–28.

[28] W. Weimer, G. C. Necula, Exceptional situations and program reliability, ACM Transactions on Programming Languages and Systems 30 (2008).

[29] G. Petrosino, E. Iotti, S. Monica, F. Bergenti, A description of the Jadescript type system, in: Proceedings of the 3rd International Conference on Distributed Artificial Intelligence (DAI 2022), volume 13170 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 206–220.

[30] FIPA, FIPA Agent Management Specification, www.fipa.org/specs/fipa00023/SC00023J.html, 2002. Accessed on June 24th, 2021.

[31] E. Iotti, G. Petrosino, S. Monica, F. Bergenti, Two agent-oriented programming approaches checked against a coordination problem, in: Proceedings of the 2020 International Symposium on Distributed Computing and Artificial Intelligence, Springer, 2020, pp. 60–70.

[32] E. Iotti, G. Petrosino, S. Monica, F. Bergenti, Exploratory experiments on programming autonomous robots in Jadescript, in: Proceedings of the 1st Workshop on Agents and Robots for Reliable Engineered Autonomy (AREA 2020) at the European Conference on Artificial Intelligence (ECAI 2020), volume 319 of *Electronic Proceedings in Theoretical Computer Science*, UNSW, 2020, pp. 55–67.

[33] S. L. Graham, S. P. Rhodes, Practical Syntactic Error Recovery, Communications of the ACM 18 (1975).

[34] F. Bellifemine, G. Caire, D. Greenwood, Developing Multi-Agent Systems with JADE, Wiley Series in Agent Technology, John Wiley & Sons, 2007.